SPECIFICATION

On page 2 of the original specification, please replace the first paragraph, which begins with the phrase, "Further, it is difficult to encode", with the following replacement paragraph:

Further, it is difficult to encode contextual information in the misuse model. This leads to a type of false positive detection in the sense that an attack against a particular architecture or operating system is detected, but the target is neither of these, thereby rendering the attack benign. Many misuse models do not provide suitable syntax for associating certain signatures with specific targets. In addition, systems that implement the misuse model typically only provide basic pattern matching features, such that an attacker may be able to bypass an [[DS]]IDS by obfuscating a known attack slightly so that it no longer matches the signature. This highlights a major drawback associated with the misuse model, namely that signatures are stored and matched against in a syntactic manner.

On page 2 of the original specification, please replace the second paragraph, which begins with the phrase, "The other common intrusion detection paradigm", with the following replacement paragraph:

The other common intrusion detection paradigm is known as anomaly detection. In this model, the system has knowledge of normal behaviour and looks for exceptional activity. Normal behaviour is defined in terms of metrics recorded when the system is running routinely. If the IDS is concerned with specific user activity then the set of metrics may include the number of processes running and the number of files open, whereas if the [[UDS]]IDS is monitoring a system as a whole, the set of metrics may include CPU usage, the number of network connections open, and sequences of system calls. The main advantage of the anomaly-based model is the low proportion of false negative detections made, which can be reduced further by increasing the sensitivity to changes in the metrics. Unlike the misuse model, this model can potentially identify previously unseen attacks. In addition, once the system has been trained, it requires less human intervention than the misuse model, because it can automatically suggest updates to its knowledge (based on previous system activity). However, this substantially increases the threat of subversion errors. Further, an attacker may even be able to weaken the system from the start by attempting intrusions whilst the system is learning normal behaviour.

On page 3 of the original specification, please replace the second paragraph, which begins with the phrase, "In the anomaly-based model", with the following replacement paragraph:

In the anomaly-based model, a much more autonomous form of learning takes place, in the sense that human input is required to provide basic knowledge as the starting point for the system. From here, the system then updates its knowledge base according to its own criteria. However, in view of the lack of human intervention during the learning and knowledge base updating process, and the fact that this learning process is based on the [[user s]]user's initial intentions (which may change over time depending on activities occurring on the network), it is relatively easy for a potential attacker to influence this process, resulting in the occurrence of an unacceptable number of subversion errors.

On page 5 of the original specification, please replace the third paragraph, which begins with the phrase, "In one embodiment", with the following replacement paragraph:

In one embodiment, the present invention extends the misuse intrusion detection paradigm (described above) to create an IDS that searches computer input traffic in a semantic matching manner to determine the contextual function of the traffic, as opposed to simple signature matching of known sinister traffic (or syntactic matching). Representing the knowledge base of matching rules in a logic programming language, in a preferred embodiment, allows for greater flexibility and for combination with other types of information (e.g. contextual information about the host computers that the [[DS]]IDS is attempting to protect).

On page 6 of the original specification, please replace the second paragraph, which begins with the phrase, "The second aspect of the invention in particular", with the following replacement paragraph:

The second aspect of the invention in particular is a process by which the IDS can automatically improve the protection it offers by learning new protection rules. These new rules are learnt (in a preferred embodiment) by applying ILP to attacks that breach, or nearly breach, to the existing [[DDS]]IDS.

On page 7 of the original specification, please replace the fifth paragraph, which begins with the phrase, "The following description presents a basic knowledge", with the following replacement paragraph:

The following description presents a basic knowledge base for an intrusion detection system according to an exemplary embodiment of the present invention. The described knowledge base contains information about a single class of security flaw, the input validation error, present in most vulnerability taxonomies, whereby the word taxonomy may be defined as "the science or practice of classification", and a vulnerability may be defined as "a feature or bug in a system or program which enables an attacker to bypass security measures".

On page 7 of the original specification, please replace the sixth paragraph, which begins with the phrase, "One known class of vulnerability", with the following replacement paragraph:

One known class of vulnerability relates to incomplete input validation; the most common form of which is the buffer overflow. The circumstances in which a buffer overflow can occur are well understood in the art and wholly avoidable if the software is developed according to one of many secure programming checklists. Although known modifications to develop [[and]]runtime environments have been proposed in the past, which modifications are designed to detect vulnerable code and constrain buffer overflow attacks, few such systems are in widespread use, often due to the high overheads associated with them, and without runtime measures to protect against buffer overflow attacks, the need for intrusion detection is increased.

On page 8 of the original specification, please replace the fourth paragraph, which begins with the phrase, "After the unbounded function call", with the following replacement paragraph:

After the unbounded function call, the exploit code has been transferred to the space allocated for the dangerous ~~function s~~function's local variables and the register save area. The address of the start of the exploit code has overwritten the return address. Once the dangerous function has completed, values from the (overwritten) register save area are popped off the stack back into system registers and execution continues from the new return address, executing the ~~attacker s~~attacker's code.

At the bottom of page 8 and continuing into page 9 of the original specification, please replace the last paragraph on page 8, which begins with the phrase, "where the stack for each process starts", with the following replacement paragraph:

where the stack for each process starts (fixed for each operating system/kern[[a]]el) and approximately how many bytes have been pushed onto the stack prior to the attack

On page 9 of the original specification, please replace the fifth paragraph, which is enumerated as "2." and begins with the phrase, "Determine any constrains", with the following replacement paragraph:

2. Determine any ~~constrains~~constraints that narrow the domain (often inherent to the particular kind of attack).

On page 11 of the original specification, please replace the fourth paragraph, which begins with the phrase, "The previous constraints can be classified", with the following replacement paragraph:

The previous constraints can be classified as hard constraints since they specify what must and what must not occur in the input. It is also possible to define soft constraints, that is, conditions that can only be stated in a fuzzy manner. An example of a soft constraint concerns the length of the exploit code. As can be seen from the stack [[of]]diagrams, there is a limited amount of space for the exploit code to occupy before the return address is reached. If the exploit code is larger, the return address will be set to the value represented by the four bytes of instruction code that happen to have been copied there and the attack will fail (the vulnerable software will almost certainly crash but the attacker will not accomplish his goal). A length constraint therefore states that the size of the exploit code must be smaller than the amount of memory before the return address. Of course, the latter may only be estimated and so this constraint can be paraphrased as stating that the attacker must write [[to]]the exploit code to be as compact as possible.

On page 11 and continuing into page 12 of the original specification, please replace the last paragraph on page 11, which begins with the phrase, "A final soft constraint states that the exploit code", with the following replacement paragraph:

A final soft constraint states that the exploit code is likely to be prefixed by a number of instructions that have no purpose except to idle the processor. Idle instructions are used to

reduce the uncertainty of guessing the correct return address. If there are no idle instructions the return address must be set to the beginning of the exploit code; this requires a perfect knowledge of the state of the process stack. By including a sequence of idle instructions at the start of the exploit code, the return address need only be set to an address within this sequence; this greatly improves the chances of an exploit succeeding. This sequence of idle instructions is often referred to [[a]]a NOP sledge (NOP is the opcode that represents a no operation instruction within the CPU). FIG. 4 shows a more realistic layout of the stack after an unbounded function call has completed containing a NOP sledge and multiple copies of the return address. The number of idle instructions that prefix the exploit code may be variable depending on the ~~attacker s~~attacker's knowledge of the layout of the stack. Early buffer overflow exploits only used the NOP opcode in creating a sequence of idle instructions. This, however, is easily detectable by pattern matching. Recent exploits use instructions that are deemed to be idle within the context of the exploit. An example of an idle instruction therefore, might be some arithmetic operation on a register which is not referenced in the exploit code. The attacker may also decide to use multi-byte instructions (i.e., the opcode and operand take up more than a single byte) as idle instructions. This reduces the chances of the exploit working (since the return address must now contain the address of the first byte of a multi-byte instruction) but if the circumstances are such that the attack can be repeated multiple times then this method can be successful.

On page 12 of the original specification, please replace the second paragraph, which appears under the heading of "Summary of Constraints", and which begins with the phrase, "due to the uncertain constraint", with the following replacement paragraph:

Due[[due]] to the ~~uncertain constraint~~uncertainty, there will be a sequence of idle instructions.

On page 12 of the original specification, please replace the third paragraph, which begins with the phrase, "as there can be no hard-coded address", with the following replacement paragraph:

As[[as]] there can be no hard-coded address, there will be code to indirectly load the address of the shell string into a register; this code will be near the start of the exploit code.

On page 12 of the original specification, please replace the fourth paragraph, which is enumerated as "·1", and which begins with the phrase, "due the Linux method", with the following replacement paragraph:

[[·1]]1. Due[[due]] to the Linux method of executing a system call, there will be code to set up the parameters then jump into kernel mode; this will make up the last few instructions in the shellcode.

On page 12 of the original specification, please replace the fifth paragraph, which is enumerated as "·2", and which begins with the phrase, "code prior to the setting up of the system call", with the following replacement paragraph:

[[·2]]2. Code[[code]] prior to the setting up of the system call [[with]]will null-terminate the sequence of characters representing the shell; this involves sub-steps of generating a zero byte inside a register and moving the zero byte to the address of the end of the sequence of characters.

On page 19 of the original specification, please replace the second paragraph, which begins with the phrase, "The proposed active IDS blurs the boundary", with the following replacement paragraph:

The proposed active IDS blurs the boundary between the role of the firewall (a system designed to prevent unauthorized access to or from a private network), and the intrusion detection system. Most firewall rule sets allow reasonably complex filters to be defined based on packet attributes (e.g. source and destination address, port and the ~~packet s~~packet's flags) but few provide anything more than basic pattern matching on the payload of the packet (this is known as content filtering). The Prolog IDS allows for semantic matching as discussed above as well as representation of contextual information (e.g. what operating system a certain host is running and what patches have been applied). If the IDS were to be deployed as a form of firewall then packet information could be represented very easily, since Prolog is well suited to modelling systems of related objects, each of which has a set of attributes. This allows the administrator to define intrusion detection predicates that combine advanced content matching, contextual information and packet attribute data. This type of IDS could be implemented in a Prolog system that provides an interface for a common programming language such as C (to allow for low level network access to read in the packet) and a bridge to allow for interaction with

relational databases (to manipulate each packet s set of attributes). Checking for computer virii is analogous to detecting intrusions. Most virus checkers implement the misuse model; that is, they have a signature for each virus and perform pattern matching on the file. Some virus checkers offer heuristic scanning (i.e. Norton Antivirus). Files are assigned a probability that they are harmful based on certain characteristics. Heuristic scanning has had particular success with macro-virii (a malicious macro embedded in a document) and script virii for the Windows platform (typically transferred by email clients). It is more difficult to determine whether an arbitrary executable file is malicious since its structure is more complex. An extension to the second aspect of the invention would be a hostile code analy[[s]]zer, written in Prolog, that is able to improve its performance using ILP techniques (utli[[s]]zing the third aspect of the invention).